Your COMPUTER COURSE

AUGUST 1986

A COMPREHENSIVE GUIDE TO PROGRAMMING – PART 6

FINAL PART

ARE YOU A COMPUTER MASTERMIND?
FIND OUT WITH OUR COMPUTER GAME

A COMPUTER EXCUSE ME
USING INTERRUPTS EFFECTIVELY

WHERE TO GO FROM HERE
OUR GUIDE TO FURTHER READING

AND SO TO Z
. . . CONCLUDING OUR GUIDE TO COMPUTER JARGON

In common with educational institutions throughout the country, the *Your Computer* Course is having a break for the summer. Unlike other courses, there is no examination at the end of our introduction to the techniques of programming your microcomputer. We hope, though, that you will take the opportunity to build on the knowledge gained during the last few months, honing your programming skills by developing your own software. To help you find additional sources of information, we have included in this final part of the course a guide to additional reading in which we list some advanced texts on programming techniques for the leading computers.

Also included in this month's concluding instalment of the course is the final part of John Lettice's glossary, in which he reveals which computer jargon terms begin with the letter Z.

In addition, we assess the Fortran language and list a simple game based on the *Mastermind* game so popular in the late 1970s.

In conclusion, we hope that you have enjoyed the course and learned from it. Do not rest on your laurels. Continue to practise your programming skills during the summer months and watch for an advanced *Your Computer* programming guide in the future.

INDEX

Techniques 42

Glossary (T-Z) 43

Programming 44

Interrupts 46

Books 48

Contributors:

John Lettice

Geof Wheelwright

David Janna

Making the most of your computer

It is often possible to save a good deal of time when programming by allowing the computer's hardware to do some of the work for you. Yet many people still fail to make use of all a computer's powers when solving their problems. Alexander Graham shows that there is more than one way to skin a cat.

There is the tale of the mechanical engineer, physicist and theoretical mathematician who are stranded in the middle of an arid plain with only a can of tinned fruit for refreshment. Their problem is that none of them has a can opener and each is asked to devise a solution to their plight.

The physicist calculates that, using the lens of his spectacles, he could focus the energy of the rays of the sun into a beam which would cut through the metal of the can and thus gain access to the fruit. The mechanical engineer, in turn, demonstrates, with a crude diagram, that by making use of available materials he could construct a primitive can opener with a sharpened rock as its cutting edge.

Those two worthies then turn to the theoretical mathematician to hear his solution. After a few moments of contemplation, he leans back and states that they should assume the tin is open and eat the contents.

That tale, aside from poking fun at the idealistic view of certain members of academic circles, is designed to show that for any given problem there are a number of solutions. Some of them will be of equal value, while others will be of little practical use. If at this stage you are wondering what it has to do with programming your home computer, remember that programming concerns the development of software to achieve a certain specified goal – in other words to solve a problem.

The programmer has at his disposal a variety of tools, not least of which is the environment offered by the operating system of the computer and the facilities provided by the programming language in use. Many programmers, though, fail to make use of another important aid to their work, the hardware of the computer on which they are working.

If a program is concerned solely with the manipulation of data input via the keyboard, with the results of the computation being displayed as text-based output on-screen, it is not likely that they will need to use any special facilities offered by the hardware of a system. Few programs are as limited in their function as that, since there is often a requirement to present data in a graphical form and, when programming games, to generate a sound output.

To produce graphics and sound, a programmer has a choice between accessing particular areas of a computer directly to achieve the desired result or making use of various facilities which may be built into the hardware of the system to achieve a similar result. In the former case, the programmer can often realise a greater control over the performance of the system.

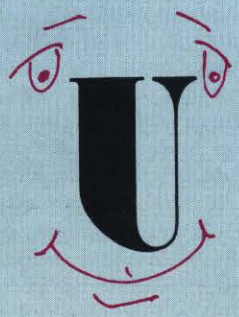
The penalty to be paid is in the programming effort to achieve the desired result. In the latter case, while the programmer may be more restricted in the control which can be exercised over the system, the programming effort will be much reduced, the designers of the system having taken some of the burden of system control off the shoulders of the programmer.

Although at times it may be hard to believe, computer manufacturers often design their machines with special facilities built in to assist programmers in achieving their goals.

To sum up, programmers should not be blinded by the joys of 'pure' programming but should remember that in many cases permitting the hardware do some of the hard work will make their lives far easier. It is a little like asking why if you have bought a dog you should do the barking. If your computer has a video control chip which can handle screen output adequately in a particular application, why re-invent the wheel by writing your own screen-driver? The question may seem trivial but it is surprising how many programmers do that; you will be a far more efficient programmer if you avoid such temptation.

Your COMPUTER COURSE

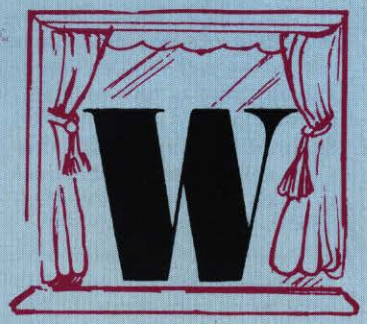
■ **Table:** A method of storing data items in memory. See **array**.
■ **Temporary storage:** Memory locations used for short-term storage of partially-completed calculations.
■ **Terminal:** Device used by the programmer to access a large computer. Terminals, like programmers, can be dumb or smart.
■ **Text editor:** The simplest, cheapest and sometimes most effective form of word processor.
■ **Toggle:** A key used as a simple switch to enable and then disable a feature of a program.
■ **Top-down programming:** Snob programming involving getting the overall concept correct and then writing subroutines to do the bits into which you have broken it down.
■ **TOS:** Operating system on the Atari ST. Atari now denies that it stands for Tramiel Operating System but we know different.
■ **Tramiel:** Almost as famous as Sir Clive Sinclair.



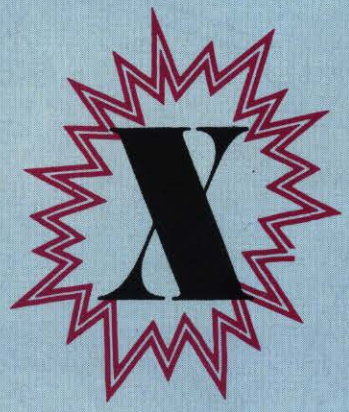
■ **Unix:** Language which is always on the point of making it but never does.

■ **Upload:** To transfer data into a computer.
■ **User-friendly:** Term used to describe a system which pretends to be easy to use until you become careless.
■ **Utility:** A program which is useful. Uncommon.

■ **Validation:** A test of the value of a data item.
■ **Variable:** A name assigned to a value which can change as the program runs.
■ **Vector:** A pointer to an address.
■ **Verify:** To test a transmitted or saved piece of datas for error.
■ **Virtual memory:** RAM expansion card undergoing design. Alternatively, programs written to work with machines with varying amounts of RAM will often use disc accesses as virtual memory to compensate for any deficiency.



■ **Wait state:** The pause most processors are compelled to take to wait for the slower memory to catch-up.
■ **Winchester:** A fixed, sealed disc unit of large capacity.
■ **Window:** An area of program or screen which can be viewed and manipulated independently.
■ **Word-wrap:** Facility in word processor which avoids words breaking across lines.



■ **X-Men:** A few comapnies, such as X-Data and Xcell, begin with X because there is not much else happening in this part of the alphabet. As they are not sufficiently important to mention here, we will have an irrelevant entry from Marvel Comics instead.

■ **Y:** Probably the least-used letter in computer programs and also short for Yes in many adventure games.
■ **Y-axis:** The horizontal part of a graph, at a right angle to the x-axis.

■ **Zero:** Like X, there is nothing much happening under Z.
■ **Zero wait state:** Because of differences between the speed at which RAM operates and the speed of the computer clock, execution of a program often has to wait while the RAM catches-up. Zero wait state memory therefore speeds execution by avoiding this.
■ **Zilog:** Manufacturer of the Z-80 processor, which would now be defunct were it not for Amstrad.

Finding the language for your needs

Fortran is not a language normally associated with microcomputers, yet versions of it are available for machines as diverse as the BBC micro, the Sinclair QL and the Atari ST. The main reason for the healthy number of home micro editions of Fortran is that it remains a popular teaching language at colleges, universities and polytechnics, where they use the kind of mainframe and minicomputer hardware best-suited to Fortran programming.

As users in some parts of the U.K. are still required to use punch cards, line printers and old mainframe systems to develop and edit Fortran, the ability to use screen, cursor keys and your micro to run a Fortran program must be a big attraction.

The question is how do you want the language to be used on your computer?

In terms of the way the language works, Fortran is a structured programming language similar in some respects to Basic. Instead of GOSUBs or procedures, however, the Fortran programmer uses what are known as a series of Do loops to accomplish the job.

Again, as in Basic, the Do loops contain all the instructions needed to carry-out each constituent part of the program. Those Do loops are then called from the beginning of the program and executed in the order in which you want them carried-out.

As computer standards have evolved, so has Fortran, with what many consider to be the penultimate version of the language being defined in 1977 as Fortran 77 – an unimaginative designation only a federation group like programmers could be relied on to conceive. Thus it is a good idea, if you are looking for a micro version of the language, to make sure that it conforms to the standard.

The alternative, of course, is to ensure that the version is at least a good superset of the standard Fortran. In too many instances, programmers become so enthusiastic about improving a standard implementation of the language that may make it entirely non-standard. It is not a problem, for example, if an edition of Fortran adds non-standard colour and sound commands which permit access to specific features of your micro, so long as they do not omit standard Fortran commands for printing information to the screen, getting it from the keyboard or handling any other machine-specific input/output.

By now you will have had a fair taste of the wide variety of languages available for most makes of micro but you also may well not be sure as to the way in which you choose the proper language for your needs. As with most things, the type of language you need depends a great deal on the type of job you want to do.

For most jobs, the Basic which is almost certain to have been included with your machine will be more than adequate but Basic has its limitations. It is often difficult, for example, to move Basic programs from one machine to another.

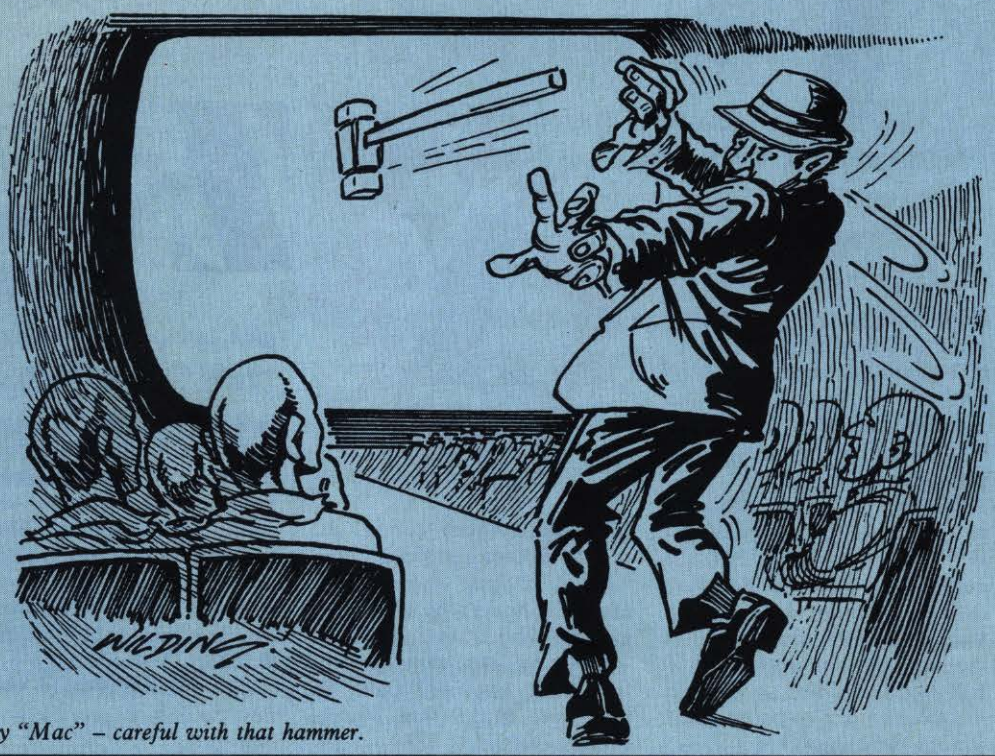
While one machine may offer procedures in its Basic, another will use GOSUBs. On the BBC micro you can define music with the sound and envelope commands, while it takes a series of pokes to do the same thing on the Commodore 64. In short, Basic is not necessarily a portable language. You would do better to look at something like C or Pascal if you require portability.

Fortran, the long-established language, is now seeing the light on some of the larger home computers.

If speed is a concern, you should consider something along the lines of Forth, written to the Forth Interest Group standard. Versions are available for most popular makes of machine and the source code should be obtainable from Forth Interest Groups if you want to do your own conversion.

You also need to address the question of how you want the language to be used on your machine. In most cases there is not a great deal of choice; the languages must be loaded from tape or disc and fit in RAM. On machines such as the BBC micro, Sinclair QL and Commodore 64, cartridge software can be used, eliminating the need to use valuable RAM space.

Cartridge-based programming languages are not, of course, modified as easily as those in RAM but they tend to operate faster and provide a great deal more workspace than RAM-based offerings.



Hey "Mac" – careful with that hammer.

Mastermind on the BBC

Geof Wheelwright looks at the problems involved in designing a strategic game.

While dealing extensively with the techniques involved in writing games software and how to incorporate animation and sound, we follow by turning to the problems involved in designing a strategic game. Computer versions of such well-known classics as *Chess* and *Reversi* require the use of analysis tools which try to make some sense of the game player's input and make a sensible response.

One of the classic studies in this type of strategic game is *Mastermind* – not the TV variety but the game popular in the late 1970s as a board game. As played by people it involves a set of six coloured markers being placed in a random order by an opponent and you trying to guess where each of the colours is located, using the fewest number of guesses.

To make the program as portable as possible, we have written ours using numbers rather than colours to play the game. That means that the computer will pick six numbers from a set of six or more numbers pre-determined by you. If, for example, you set the limit to nine, the computer will pick six numbers from the range one to nine from which to obtain the six numbers you will have to guess. Having picked the number, the computer will place the set in order and process your guesses.

To allow you to expand the program, written for the BBC micro, to include title screens and other features we have started the program numbering at 50. Here is a brief indication of how it works:

50
Sets the screen mode and declares the initial values of the variables T and U.

60 and 70
Generates the screen message telling players how to play the game and finding how difficult they want to make it. The higher the field of numbers from which they can guess the more difficult the game.

80
Generates random variables for each of the six numbers in the set.

90
Is a check against the computer guessing the same number twice. To make the game more difficult, this line could be removed.

100
The players can enter their guesses; each time they do the variable T, which counts the number of guesses which have been made, is increased by one.

120-170
Calculate the value of variable Q, which determines whether or not the number guessed is in the combination.

180-230
Check whether or not the player has guessed a number which is both correct in placement and type.

240-290
Check the variables Q, P and U to decide appropriate responses to players' input; the U variable checks to see whether a player has cheated by looking at a quick flash of numbers on-screen as a hint.

This is obviously a simple version of the classic game and will take a little more work to convert for other machines. If you want to enhance it to use on-screen colours, a routine with variables A to FF to those of the colours required would do the trick.

On-screen blocks formed from sprites could then be designed and text run along the bottom of the screen. The fundamental design of the game would, however, remain unchanged.

Mastermind

```
 50 MODE 3:T=0:U=0
 60 PRINT "How many numbers would you
like to be in the field of guesses: if y
ou answer 9, the BBC will generate six n
umbers between 1 and 9. You MUST answer
with a"
 70 PRINT " number 6 or higher because
 the computer is expecting to be able to
 generate at least six different numbers
. BE SURE TO HIT RETURN AFTER EACH GUESS
":INPUT N
 80 A=RND(N):B=RND(N):C=RND(N):D=RND(N
):E=RND(N):F=RND(N)
 90 IF A=B OR A=C OR A=D OR A=E OR A=F
 OR B=C OR B=D OR B=E OR B=F OR C=D OR C
=E OR C=F OR D=E OR D=F OR E=F THEN GOTO
 80
 100 PRINT "Put in your guess with comm
as separating each number: i.e. 6,7,8,9,
3,4. What is your guess":T=T+1:INPUT G,H
,I,J,K,L
 120 Q=0:IF A=G OR A=H OR A=I OR A=J OR
 A=K OR A=L THEN Q=Q+1
 130 IF B=G OR B=H OR B=I OR B=J OR B=K
OR B=L THEN Q=Q+1
 140 IF C=G OR C=H OR C=I OR C=J OR C=K
OR C=L THEN Q=Q+1
 150 IF D=G OR D=H OR D=I OR D=J OR D=K
OR D=L THEN Q=Q+1
 160 IF E=G OR E=H OR E=I OR E=J OR E=K
OR E=L THEN Q=Q+1
 170 IF F=G OR F=H OR F=I OR F=J OR F=K
OR F=L THEN Q=Q+1
 180 P=0:IF A=G THEN P=P+1
 190 IF B=H THEN P=P+1
 200 IF C=I THEN P=P+1
 210 IF D=J THEN P=P+1
 220 IF E=K THEN P=P+1
 230 IF F=L THEN P=P+1
 240 IF A=G AND B=H AND C=I AND D=J AND
E=K AND F=L AND U=0 THEN CLS
 250 IF A=G AND B=H AND C=I AND D=J AND
E=K AND F=L AND U>0 THEN CLS:PRINT "YOU
CHEATED ";U;" TIMES, BUT..."
 260 IF A=G AND B=H AND C=I AND D=J AND
 E=K AND F=L THEN PRINT "YOU'RE RIGHT AN
D YOU MADE IT IN ";T;" GUESSES. YOUR COR
RECT GUESS WAS..":PRINT A,B,C,D,E,F:END
ELSE PRINT "TRY AGAIN. HINT: IF YOU WANT
 TO CHEAT PRESS 'Y' IN THE NEXT 3 secs"
 270 IF INKEY$(450)="Y" THEN U=U+1:PRIN
T "THE NUMBERS ARE....";A,B,C,D,E,F ELSE
 GOTO 280
 280 PRINT "There are ";Q-P;" of your n
umbers in the combination, but in the wr
ong place"
 290 PRINT"and an additional ";P;" that
 are in the combination and in the right
 place":GOTO 100
```

Multi-tasking

If you look at the commercial games available you will notice that the action normally involves more than one object moving about the screen at the same time. Not only that, a tune may be playing in the background as well. Thinking about how that is accomplished in programming terms can lead to confusion. How do you keep track of all those events and how do you manage to do several things simultaneously?

The concept of doing more than one task at the same time is not new to computing. Many mainframe computers permit more than one person to use it at the same time. In the business micro sector, multi-tasking micros commonly allow up to five people to use the micro, each running their own applications as if they were using their own personal computers.

There are several ways in which multi-tasking is accomplished but at the CPU level it is all a matter of interrupts. Further, even though it may not be advertised as a feature, home micros are capable of multi-tasking at machine code level.

To understand what an interrupt is, imagine someone sitting at a desk typing. The telephone rings and the person stops typing, answers the call and, after finishing the call returns to the typing, continuing from where he was interrupted. An interrupt on a micro is very similar. When an interrupt occurs, the CPU stops executing the main program, attends to the interrupt and, after it has been serviced, returns back control to the main program from the point where it was interrupted.

Both the Z-80 and 6502 microprocessors produce interrupts many times each second. The operating system/kernel of the micro makes good use of those interrupts because they occur at regular intervals and thus can be used for timing purposes and so on.

Even when the micro is apparently idle, thousands of machine code instructions are being executed. The operating system of the micro is performing housekeeping operations. They include updating the screen, checking to see if any keys have been pressed, and so on. Those tasks are also done on a regular basis – typically 60 times a second – and it is possible to patch into those tasks with your own machine code routine which, in turn, will also be executed the same number of times per second.

At this stage it is worth noting the difference between foreground and background tasks. Executing a program in Basic or machine code is the foreground task. The housekeeping and any interrupt-driven routines are referred to as the background tasks.

Setting-up an interrupt routine for your micro requires you to follow some set rules which are laid down by the manufacturer of the micro. Here are details on how to set up interrupts for the Commodore 64, Amstrad and BBC micros.

Commodore 64

One of the input/output chips on the C-64 produces an interrupt every 1/60th of a second. A kernel ROM routine located at $EA31 is executed and that deals with various housekeeping tasks – see listing one.

Whenever the interrupt occurs the foregoing is executed. The processor does not pass control straight to the routine at $EA31; instead the processor looks at the address which is held at $0314/$0315 (IRQ vector) and jumps to that location. On power-up of the C-64, the address at $0314 is $EA31 and thus control is passed to the kernel routine there but as that location is in RAM it can be altered and instead the address of our routine can be inserted. Control would then pass to our routine and at the end of the routine it would be necessary to jump to $EA31 to close the loop. So, to write a simple interrupt routine three steps have to be followed – set up any necessary parameters for the routine; set the IRQ vector to point to the routine; the routine.

Assuming that the routine is to reside at $C000, a typical example may look like listing two.

The code between $C000 – $C00C sets up a sprite. The second part of the program is very important. First, interrupts are disabled. That is necessary as the address of the interrupt routine which is to be loaded into $0314 must be loaded in two steps; an interrupt just after part of the address has been loaded could prove fatal. Once $0314 has been loaded with the address of the routine and the interrupts re-enabled, the routine is being executed from that point onwards.

In the example the interrupt routine resides at $C01C and increments location $D001 which moves the sprite. Notice the crucial JMP to $EA31 which maintains the loop.

The C-64 also generates other types of interrupt, namely non-maskable interrupts and a sprite collision interrupt of which the programmer can make use. For more details consult the reference guide.

Amstrad

Amstrad users will no doubt be familiar with interrupts, as they are a major feature of Locomotive Basic. The commands AFTER and EVERY allow different Basic subroutines to be executed after a given period.

Simple interrupt handling on the Amstrad is easy. On power-

Commodore 64

```
$EA31 Update clock
$EA34 Process cursor, tape motor, scan keyboard
$EA7E Restore registers
$EA86 Exit routine
```

```
C000 LDA #$01   ; TURN ON
C002 STA $D015  ; SPRITE.
C005 LDA $#96   ; SET SPRITE 0
C007 STA $D000  ; X POS TO 150 DECIMAL.
C00A LDA #$01   ; SET SPRITE 0
C00C STA $D027  ; COLOUR TO WHITE.
                ;
C00F SEI        ; DISABLE INTERRUPTS.
C010 LDA #$1C   ; LOAD $0314 WITH LOW
C012 STA $0314  ; BYTE ADRESS OF ROUTINE.
C015 LDA #$C0   ; LOAD $0315 WITH HIGH
C017 STA $0315  ; BYTE ADDRESS OF ROUTINE.
C01A CLI        ; TURN ON INTERRUPTS.
C01B RTS        ; RETURN CONTROL TO FOREGROUND TASK.
                ;
C01C INC $D001  ; INCREASE Y POS OF SPRITE 0.
C01F JMP $EA31  ; CLOSE THE LOOP.
```

AMSTRAD

```
9E00  LD HL,#9E0E   ; LOAD #39 WITH START
9E03  LD (#39),HL   ; ADDRESS OF ROUTINE.
9E06  RET           ; RETURN TO FOREGROUND PROGRAM.
                    ;
9E07  LD HL,#B939   ; LOAD #39 WITH ADDRESS
9E0A  LD (#39),HL   ; OF IS ROUTINE.
9E0D  RET           ; RETURN TO FOREGROUND PROGRAM.
                    ;
9E0E  PUSH AF       ; SAVE REGISTER ON STACK.
9E0F  LD A,7        ; ACSII CODE FOR BELL
9E11  CALL #BB5A    ; AND MAKE THE SOUND.
9E14  POP AF        ; RESTORE REGISTER.
9E15  JP #B939      ; CONTINUE WITH LOOP.
```

up interrupt mode one (IM1) is selected. In that mode an interrupt will make a CALL to address #0038 – #38 is a vector which holds the address of the interrupt service routine which is at #B939. That vector can be amended to point to our own subroutine.

Enabling an interrupt is simple. All that is required is to load the register pair HL with the address of the interrupt routine and load the contents of HL into location #39. The end of the routine should route back to the interrupt service routine with a JP – see listing three.

In the foregoing example the code from #9E00 – #9E06 sets up the interrupt routine and the code at #9E0E will be executed every time an interrupt occurs. Note the code from #9E07 – #9E0D, which disables the interrupt by loading the interrupt service routine vector with its original value.

To enable the interrupt routine a call is made to #9E00 but nothing appears to happen. That is because the interrupt service routine becomes active only when an interrupt occurs. Try entering a line of Basic and you will hear the bell tones.

There are other ways of using interrupts on the Amstrad beside the vector at $0039. A set of pre-packaged interrupts called events are available to the user. Unlike the more crude interrupt-handling method described, they are activated only when a certain condition arises, such as the frame flyback or a key being pressed. They are more flexible to use and readers should refer to the Amstrad firmware manual for more details.

BBC

The BBC micro makes extensive use of interrupts. To facilitate writing interrupt-driven routines, the programmer can make use of events, which are a form of high-level interrupt. An event is a pre-packaged form of interrupt provided for the user's benefit – the operating system does not make use of events. An event is specific for an application. The available events on the BBC micro are output buffer empty; input buffer full; character-entering input buffer; ADC conversion complete; start of vertical sync; interval timer crossing zero; ESCAPE condition detected; RS423 error detected; Econet-generated event; user event.

Each of those conditions generates an interrupt when needed. For example, the start of vertical sync generates an interrupt 50 times a second while, on the other hand, ESCAPE condition detected will generate an interrupt only when the escape key is pressed.

It is essential to realise that even if a particular event is disabled by the programmer it continues to generate interrupts but they will not be serviced by the event-handling routine.

To understand the use of events better consider this problem. We wish to sound a beep every time the escape key is pressed. Ordinarily, using a low-level interrupt the routine would be called very 1/50th of a second regardless of whether the escape key was pressed. That would mean that the interrupt routine would first have to test to see if the escape key was pressed and, if not, return to the foreground task.

Looking at the foregoing table, we can see that there is an event just for this occasion. To use it, it must first be turned on (enabled) and the address of the interrupt routine must be loaded into the event vector (EVNTV, &220).

David Janda looks at managing interrupts while thousands of machine code instructions are being executed.

Although the ESCAPE event is enabled and the address of the interrupt routine is then patched into the event vector that does not mean that the interrupt routine is being executed. It will happen only when the ESCAPE key is pressed – see listing four.

This article has dealt only with the subject of interrupts for the more popular machines. It is for individual programmers to decide to what use they want to put the interrupts.

It is crucial that an interrupt routine does a quick execute, typically no longer than one millisecond. That may not seem to be much but at least 300 machine code instructions can be executed in that time.

It is also very important to preserve all registers when entering an interrupt routine. That is done by pushing the values held in all registers on to the stack and popping them at the end of the routine.

BBC

```
0 Output buffer empty                          5 Interval timer crossing zero
1 Input buffer full                            6 ESCAPE condition detected
2 Character entering input buffer              7 RS423 error detected
3 ADC conversion complete                      8 Econet generated event
4 Start of vertical sync                       9 User event
```

```
100   REM EVENT INTERRUPT
110   DIM MC% 100
120   FOR I=0 TO 3 STEP 3
130   P%=MC%
140   [
150   OPT I
151   .Setup     LDX #06                  \ EVENT NUMBER 6
153              LDA #14                  \ IS SELECTED BY
155              JSR &FFF4                \ CALLING OSBYTE.
160              LDA #Routine MOD 256     \ LOAD EVENT VECTOR WITH
170              STA &220                 \ LOW BYTE ADDRESS OF ROUTINE.
180              LDA #Routine DIV 256     \ LOAD EVENT VECTOR WITH
190              STA &221                 \ HIGH BYTE ADDRESS OF ROUTINE.
200              RTS                      \ RETURN TO FOREGROUND PROGRAM.
210   .Routine   PHA
220              TXA
230              PHA                      \ PRESERVE REGISTERS.
240              TYA
250              PHA
260              PHP
270              LDA #07                  \ ASCII CODE FOR BELL
280              JSR &FFEE                \ SOUND BELL.
290              PLP
300              PLA
310              TAY                      \ RESTORE REGISTERS.
320              PLA
330              TAX
340              PLA
350              RTS
360   ]
370   NEXT I
380   CALL Setup
```

Making use of the world's oldest form of medium

Geof Wheelwright looks at one of life's most reliable sources of information.

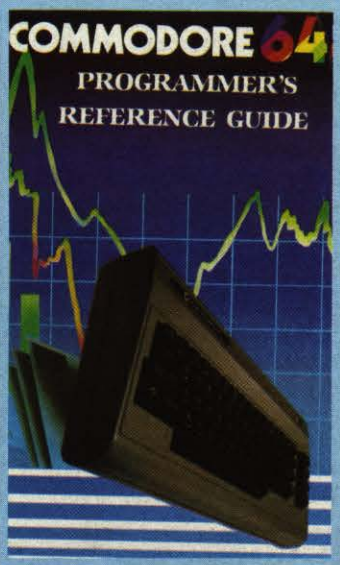
There must be some irony in the fact that the only way you can make the most of life's newest information medium, the computer screen, is by making use of one of the oldest – the printed book.

Without a good library of reference material, no programmer's workbench would be complete. The human brain has many more important things to think about than the upper memory locations of the BBC micro, or the number you have to POKE to have the colour red appear on a pixel in the middle of the Commodore 64 screen. A good set of programming guides will relieve that headache, while ensuring that the information is always at your fingertips.

One of the best examples of micro-specific programming guides is the Commodore *Programmers' Reference Guide for the 64*. It not only gives you all the necessary PEEKs, POKEs, memory locations, screen maps and other things required to unlock the full mystery of this best-selling machine but also provides a number of excellent example listings for functions such as terminal emulation, allowing you to access the RS232 port of the machine with a modem.

The guide is spiral-bound, like the excellent documentation with the BBC micro, and is thick, giving you about as much value as you are likely to obtain from a micro-specific book. It is available from Commodore or most Commodore stockists.

While most manufacturers provide advanced programming guides of this type – the Acorn *Advanced User Guide for the BBC micro* is also an excellent example of the genre – none is perhaps so essential as the one for the 64. The machine's Basic programming language means that you really need to resort to detailed PEEKs, POKEs and,



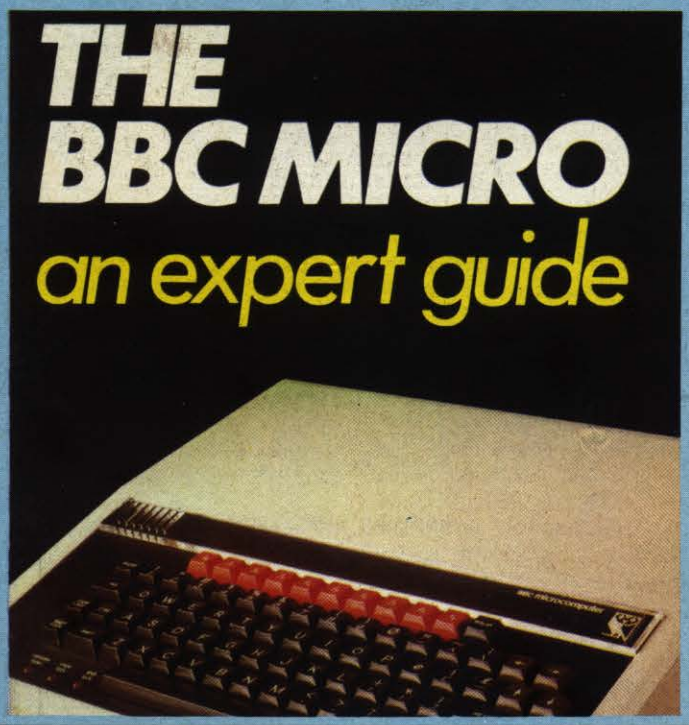
inevitably, machine code to have the 64 to do anything really impressive and useful.

Machine code programmers in general, however, would do well to buy a processor-specific rather than machine-specific guide if they are to write the most effective software and to ensure that it is portable across all machines using that processor. One of the best guides for it is Rodney Zaks' now-classic guide to programming the Zilog Z-80 chip – used in all CP/M machines, the Sinclair ZX-81, Spectrum and many others.

The *Z-80 Programmers' Guide* is published by Sybex and should be required reading for all those starting on the road to machine code hacking. Those with perhaps greater ambitions and more Z-80 experience should consider the rather long-winded *Z-80 Microprocessor Advanced Interfacing with Applications in Data Communications* by J. C. Nichols, E. A. Nichols and K. R. Musson, published by Howard W. Sams. It is a 396-page tome outlining in very heavy technical detail the vast number of applications and interfacing jobs which can be applied to the Z-80.

For those who are a little more wary of machine code than real-life languages, there is a wealth of further reading to supplement what you will have seen in that course. One of the better introductions in this area is *Computer Languages and their Uses* by Gary Marshall.

He uses comparisons with Basic to provide novice users with a point of reference for discussing other languages. The 108-page Granada book is two years old but its profiles of Basic, Prolog, Comal, Lisp and many other popular languages are as valid now as they were in 1984.

Probably the best place to seek information on more reading is a book devoted to that topic. *What to Read in Microcomputing* by C. Saiady and A. V. Stokes, published by Gower Publishing Co, is again an older book but deals with more than 400 solid, technical guides available up to that time and makes a worthwhile reference work.

It separates books into four main categories – background, languages, specific micros and hardware/software – listing the books by publisher, author and title. It is a somewhat slim volume but is worth reading if you can find it among the vast number of remaindered computer books which seem to be jamming the back rooms of many of the nation's major booksellers.